Zig-zag Sort: A Simple Deterministic Data-Oblivious Sorting Algorithm Running in O(n log n) Time

Michael T. Goodrich
Department of Computer Science
University of California, Irvine
Irvine, CA 92697 USA
goodrich@acm.org

ABSTRACT

We describe ***Zig-zag Sort***—a deterministic data-oblivious sorting algorithm running in $O(n \log n)$ time that is arguably simpler than previously known algorithms with similar properties, which are based on the AKS sorting network. Because it is data-oblivious and deterministic, Zig-zag Sort can be implemented as a simple $O(n \log n)$-size sorting network, thereby providing a solution to an open problem posed by Incerpi and Sedgewick in 1985. In addition, Zig-zag Sort is a variant of Shellsort, and is, in fact, the first deterministic Shellsort variant running in $O(n \log n)$ time. The existence of such an algorithm was posed as an open problem by Plaxton *et al.* in 1992 and also by Sedgewick in 1996. More relevant for today is the fact that the existence of a simple data-oblivious deterministic sorting algorithm running in $O(n \log n)$ time simplifies the "inner-loop" computation in several proposed oblivious-RAM simulation methods (which utilize AKS sorting networks), and this, in turn, implies simplified mechanisms for privacy-preserving data outsourcing in several cloud computing applications.

1. INTRODUCTION

An algorithm is ***data-oblivious*** if its sequence of possible memory accesses is independent of its input values. Thus, a deterministic algorithm is data-oblivious if it makes the same sequence of memory accesses for all its possible inputs of a given size, n, with the only variations being the outputs of atomic primitive operations that are performed. For example, a data-oblivious sorting algorithm may make "black-box" use of a ***compare-exchange*** operation, which is given an ordered pair of two input values, (x, y), and returns (x, y) if $x \leq y$ and returns (y, x) otherwise. A sorting algorithm that uses only compare-exchange operations is also known as a ***sorting network*** (e.g., see [4,16]), since it can be viewed as a pipelined sequence of compare-exchange gates performed on pairs of n input wires, each of which is initially provided with an input item. The study of data-oblivious sorting networks is classic in algorithm design, including such vintage methods as bubble sort, Batcher's odd-even and bitonic sorting networks [5], and the AKS sorting network [1, 2] and its variations [6, 17, 20, 21, 27]. In addition, many versions of Shellsort (e.g., see [26]) are data-oblivious sorting algorithms, which trace their origins to a classic 1959 paper by the algorithm's namesake [28]. More recently, examples of randomized data-oblivious sorting algorithms running in $O(n \log n)$ time that sort with high probability include constructions by Goodrich [10, 11] and Leighton and Plaxton [18].

One complicating feature of previous work on deterministic data-oblivious sorting is that all known algorithms running in $O(n \log n)$ time [1, 2, 6, 17, 20, 21, 27] (which are based on the AKS sorting network) are arguably conceptually complex, while many of the known algorithms running in $\omega(n \log n)$ time are conceptually simple. For instance, given an unsorted array, A, of n comparable items, Batcher's odd-even and bitonic sorting networks [5], and the Shellsort implementation by Pratt [25], are based on the simple approach of performing a sequence of $O(n \log^2 n)$ compare-exchange operations between pairs of items stored at obliviously-defined index intervals. As with Shellsort-based algorithms (e.g., see [26]), the compare-exchanges in these cases are initially between pairs that are far apart in A and the distances between such pairs are gradually reduced over the course of the algorithm until one is certain that A is sorted. In terms of asymptotic performance, the best previous Shellsort variant is due to Pratt [25], which runs in $\Theta(n \log^2 n)$ time and is based on the elegant idea of comparing pairs of items separated by intervals that determined by a monotonic sequence of the products of powers of 2 and 3 less than n. There has subsequently been a considerable amount of work on the Shellsort algorithm [28] since its publication over 50 years ago (e.g., see [26]), but none of this previous work has led to a simple deterministic data-oblivious sorting algorithm running in $O(n \log n)$ time.

1.1 Privacy-Preserving Data Storage in Cloud Computing Applications

Independent of their historical appeal, data-oblivious algorithms are having a resurgence of interest of late, due to their applications to privacy-preserving cloud computing. In such applications, a client, Alice, outsources her data to an honest-but-curious server, Bob, who processes read/write requests for Alice. In order to protect her privacy, Alice must both encrypt her data and obfuscate any data-dependent access patterns for her data. Fortunately, she can achieve these two goals through any of a number of recent results for simulating arbitrary RAM algorithms in a privacy-preserving manner in a cloud-computing environment using data-oblivious sorting as an "inner-loop" computation (e.g., see [8, 9, 12, 13]). A modern challenge, however, is that many of these simulation results either use the AKS sorting network for this inner loop or compromise on asymptotic performance.

STOC '14 May 31 – June 30 2014, New York, NY USA

ACM 978-1-4503-2710-7/14/05 ...$15.00.
http://dx.doi.org//10.1145/2591796.2591830.

1.2 Our Results

We provide a simple deterministic data-oblivious sorting algorithm running in $O(n \log n)$ time, which we call ***Zig-zag Sort***. This result solves the well-known (but admittedly vague) open problem of designing a "simple" sorting network of size $O(n \log n)$, posed by Incerpi and Sedgewick [15]. Zig-zag Sort is a variant of Shellsort, and is, in fact, the first deterministic variant of Shellsort running in $O(n \log n)$ time, which also solves open problems of Sedgewick [26] and Plaxton *et al.* [22, 23].

Shellsort is actually a family of sorting algorithms, all having a similar structure (e.g., see the book chapter by Sedgewick [26]), which we generalize slightly here. A ***Shellsort algorithm*** is defined in terms of a sequence of positive integers, $(h_1, h_2, \ldots, h_k)$, which is known as its ***increment sequence***. The sorting of an array, A, of size n, is done by a series of rounds, where round r, for $r = 1, 2, \ldots, k$, comprises a sequence of compare-exchange operations of the form CompareExchange$(A[i], A[j])$, such that

$$i - h_r \leq j \leq i + h_r.$$

We say that a Shellsort algorithm is ***rigid*** if each such compare-exchange operation is restricted to be between a cell at index i and a cell at index $j = i \pm h_r$, and round r is required to result in each subsequence of elements, $(A[i], A[i + h_r], A[i + 2h_r], \ldots)$, stored in cells with indices at intervals of length h_r, being in sorted order.

Historically, the previous deterministic Shellsort algorithms are rigid (e.g., see [26]), whereas previous randomized Shellsort algorithms are nonrigid [10, 11]. Zig-zag Sort is a Shellsort algorithm using the increment sequence $(n, n/2, n/4, n/8, \ldots, 2, 1)$, where n is a power of 2, and it differs from previous deterministic Shellsort variants in that it is nonrigid. As it turns out, such a Shellsort algorithm must necessarily be nonrigid in order to achieve an $O(n \log n)$ running time, since any rigid Shellsort algorithm must have a running time of at least $\Omega(n \log^2 n/(\log \log n)^2)$, and any such algorithm with a monotonically decreasing increment sequence must have a running time that is $\Omega(n \log^2 n/ \log \log n)$, according to known lower bounds [7, 22–24].

In this paper, we concentrate primarily on conceptual simplicity, with the result that the constant factors in our analysis of Zig-zag Sort are admittedly not small. These constant factors are nevertheless smaller than those for constructive versions of the AKS sorting network [1, 2] and its recent optimization by Seiferas [27], and are on par with the best non-constructive variants of the AKS sorting network [6, 20, 21]. Thus, for several oblivious-RAM simulation methods (e.g., see [8, 9, 12, 13]), Zig-zag Sort provides a conceptually simple alternative to the previous $O(n \log n)$-time deterministic data-oblivious sorting algorithms, which are based on the AKS sorting network.[1] The conceptual simplicity of Zig-zag Sort is not matched by a simplicity in proving it is correct, however. Instead, its proof of correctness is based on a fairly intricate analysis involving the tuning of several parameters with respect to a family of potential functions. Thus, while the Zig-zag Sort algorithm can be described in a few dozen lines of pseudocode, our proof of correctness consumes much of this paper, with most of the details relegated to an appendix and Arxiv version of this paper[2].

2. THE ZIG-ZAG SORT ALGORITHM

The Zig-zag Sort algorithm is based on repeated use of a procedure known as an ***ϵ-halver*** [1–3, 19], which incidentally also forms the basis for the AKS sorting network and its variants.

- An ***ϵ-halver*** is a data-oblivious procedure that takes a pair, (A, B), of arrays of comparable items, with each array being of size n, and performs a sequence of compare-exchanges, such that, for any $k \leq n$, at most ϵk of the largest k elements of $A \cup B$ will be in A and at most ϵk of the smallest k elements of $A \cup B$ will be in B, where $\epsilon \geq 0$.

In addition, there is a relaxation of this definition, which is known as an ***(ϵ, λ)-halver*** [3]:

- An ***(ϵ, λ)-halver*** satisfies the above definition for being an ϵ-halver for $k \leq \lambda n$, where $0 < \lambda < 1$.

We introduce a new construct, which we call a ***(δ, λ)-attenuator***, which takes this concept further:

- A ***(δ, λ)-attenuator*** is a data-oblivious procedure that takes a pair, (A, B), of arrays of comparable items, with each array being of size n, such that k_1 of the largest k elements of $A \cup B$ are in A and k_2 of the smallest k elements of $A \cup B$ are in B, and performs a sequence of compare-exchanges such that at most δk_1 of the largest k elements will be in A and at most δk_2 of the smallest k elements will be in B, for $k \leq \lambda n$, $0 < \lambda < 1$, and $\delta \geq 0$.

We give a pseudo-code description of Zig-zag Sort in Figure 1. The name "Zig-zag Sort" is derived from two places that involve procedures that could be called "zig-zags." The first is in the computations performed in the outer loops, where we make a Shellsort-style pass up a partitioning of the input array into subarrays (in what we call the "outer zig" phase) that we follow with a Shellsort-style pass down the sequence of subarrays (in what we call the "outer zag" phase). The second place is inside each such loop, where we preface the set of compare-exchanges for each pair of consecutive subarrays by first swapping the elements in the two subarrays, in a step we call the "inner zig-zag" step. Of course, such a swapping inverts the ordering of the elements in these two subarrays, which were presumably put into nearly sorted order in the previous iteration. Nevertheless, in spite of the counter-intuitive nature of this inner zig-zag step, we show in the analysis section below that this step is, in fact, quite useful.

Algorithm ZigZagSort(A)

```
1:  A_1^(0) ← A
2:  for j ← 1 to k do
3:      for i ← 1 to 2^(j−1) do {splitting step}
4:          Partition A_i^(j−1) into halves, defining subarrays, A_(2i−1)^(j)
            and A_(2i)^(j), of size n/2^j each
5:          Reduce(A_(2i−1)^(j), A_(2i)^(j))
6:      for i ← 1 to 2^j − 1 do {outer zig}
7:          Swap the items in A_i^(j) and A_(i+1)^(j)      {inner zig-zag}
8:          Reduce(A_i^(j), A_(i+1)^(j))
9:      for i ← 2^j downto 2 do {outer zag}
10:         Swap the items in A_i^(j) and A_(i−1)^(j)      {inner zig-zag}
11:         Reduce(A_(i−1)^(j), A_i^(j))
```

Figure 1: Zig-zag Sort (where $n = 2^k$). The algorithm, Reduce(A, B), is simultaneously an ϵ-halver, a $(\beta, 5/6)$-halver, and a $(\delta, 5/6)$-attenuator, for appropriate values of ϵ, δ, and β. Assuming that Reduce runs in $O(n)$ time, Zig-zag Sort clearly runs in $O(n \log n)$ time.

We illustrate, in Figure 2, how an outer zig phase would look as a sorting network.

[1] We stress, however, that Zig-zag Sort is ***not*** a parallel algorithm, like AKS, which has $O(\log n)$ depth. Even when Zig-zag Sort is run in parallel as a sorting network, it still runs in $O(n \log n)$ time.
[2] http://arxiv.org/abs/1403.2777

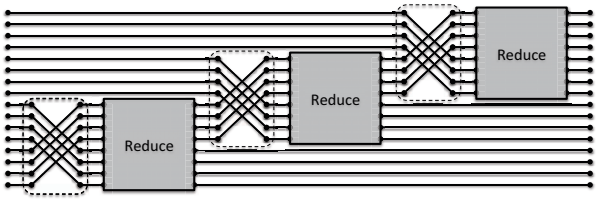

Figure 2: An outer zig phase drawn as a sorting network, for $j = 2$ and $n = 16$. The inner zig-zag step is shown inside a dashed rounded rectangle. Note: the inner zig-zag step could alternatively be implemented as a compare-exchange of each element in the lower half with a unique element of the upper half; we implement it as a swap, however, to reduce the total number of comparisons.

3. HALVERS AND ATTENUATORS

In this section, we give the details for Reduce, which is simultaneously an ϵ-halver, $(\beta, 5/6)$-halver, and $(\delta, 5/6)$-attenuator, where the parameters, ϵ, β, and δ, are functions of a single input parameter, $\alpha > 0$, determined in the analysis section (§4) of this paper. In particular, let us assume that we have a linear-time α-halver procedure, Halver, which operates on a pair of equal-sized arrays whose size is a power of 2. There are several published results for constructing such procedures (e.g., see [14, 29]), so we assume the use of one of these algorithms. The algorithm, Reduce, involves a call to this Halver procedure and then to a recursive algorithm, Attenuate, which makes additional calls to Halver. See Figure 3.

Algorithm Attenuate(A, B):
1: **if** $n \leq 8$ **then**
2: Sort $A \cup B$ and **return**
3: Partition A into halves, defining $A_1^{(1)}$ and $A_2^{(1)}$, and partition B into halves, defining $B_1^{(1)}$ and $B_2^{(1)}$
4: Halver$(A_1^{(1)}, A_2^{(1)})$
5: Halver$(B_1^{(1)}, B_2^{(1)})$
6: Halver$(A_2^{(1)}, B_1^{(1)})$
7: Attenuate$(A_2^{(1)}, B_1^{(1)})$ {first recursive call}
8: Partition $A_2^{(1)}$ into halves, defining $A_1^{(2)}$ and $A_2^{(2)}$, and partition $B_1^{(1)}$ into halves, defining $B_1^{(2)}$ and $B_2^{(2)}$
9: Halver$(A_1^{(2)}, A_2^{(2)})$
10: Halver$(B_1^{(2)}, B_2^{(2)})$
11: Halver$(A_2^{(2)}, B_1^{(2)})$
12: Attenuate$(A_2^{(2)}, B_1^{(2)})$ {second recursive call}

Algorithm Reduce(A, B):
1: **if** $n \leq 8$ **then**
2: Sort $A \cup B$ and **return**
3: Halver(A, B)
4: Attenuate(A, B)

Figure 3: The Attenuate and Reduce algorithms. We assume the existence of an $O(n)$-time data-oblivious procedure, Halver(C, D), which performs an α-halver operation on two subarrays, C and D, each of the same power-of-2 size. We also use a partition operation, which is just a way of viewing a subarray, E, as two subarrays, F and G, where F is the first half of E and G is the second half of E.

We illustrate the data flow for Attenuate in Figure 4.

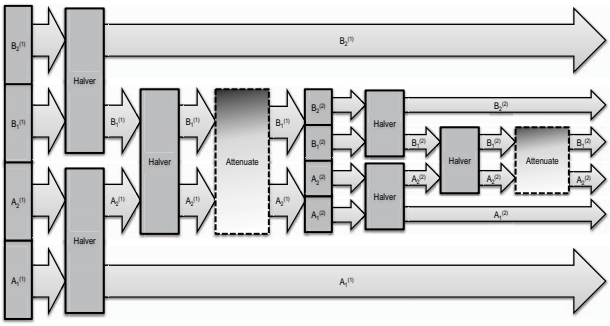

Figure 4: Data flow in the Attenuate algorithm.

4. AN ANALYSIS OF ZIG-ZAG SORT

Modulo the construction of a linear-time α-halver procedure, Halver, which we discuss in more detail in Section 5, the above discussion is a complete description of the Zig-zag Sort algorithm. Note, therefore, that the Reduce algorithm runs in $O(n)$ time, since the running time for the general case of the recursive algorithm, Attenuate, can be characterized by the recurrence equation,

$$T(n) = T(n/2) + T(n/4) + bn,$$

for some constant $b \geq 1$. In terms of the running time of Zig-zag Sort, then, it should be clear from the above description that the Zig-zag Sort algorithm runs in $O(n \log n)$ time, since it performs $O(\log n)$ iterations, with each iteration requiring $O(n)$ time. Proving that Zig-zag Sort is correct is less obvious, however, and doing so consumes the bulk of the remainder of this paper.

4.1 The 0-1 Principle

As is common in the analysis of sorting networks (e.g., see [4, 16]), our proof of correctness makes use of a well-known concept known as the ***0-1 principle***.

THEOREM 1 (**THE 0-1 PRINCIPLE** [4, 16]). *A deterministic data-oblivious (comparison-based) sorting algorithm correctly sorts any input array iff it correctly sorts a binary array of 0's and 1's.*

Thus, for the remainder of our proof of correctness, let us assume we are operating on items whose keys are either 0 or 1. For instance, we use this principle in the following lemma, which we use repeatedly in our analysis, since there are several points when we reason about the effects of an ϵ-halver in contexts beyond its normal limits.

LEMMA 2 (**OVERFLOW LEMMA**). *Suppose an ϵ-halver is applied to two arrays, A and B, of size n each, and let a parameter, $k > n$, be given. Then at most $\epsilon n + (1-\epsilon)\cdot(k-n)$ of the k largest elements in $A \cup B$ will be in A and at most $\epsilon n + (1-\epsilon)\cdot(k-n)$ of the k smallest elements in $A \cup B$ will be in B.*

PROOF. Let us focus on the bound for the k largest elements, as the argument for the k smallest is similar. By the 0-1 principle, suppose A and B are binary arrays, and there are k 1's and $2n-k$ 0's in $A \cup B$. Since $2n - k < n$, in this case, after performing an ϵ-halver operation, at most $\epsilon(2n-k)$ of the 0's will remain in B. That is, the number of 1's in B is at least $n - \epsilon(2n-k)$, which implies that the number of 1's in A is at most

$$\begin{aligned} k - (n - \epsilon(2n-k)) &= k - n + 2\epsilon n - \epsilon k \\ &= \epsilon n + (1-\epsilon)\cdot(k-n). \end{aligned}$$

□

Because of the 0-1 principle, we can characterize the distance of a subarray from being sorted by counting the number of 0's and 1's it contains. Specifically, we define the ***dirtiness***, $D(A_i^{(j)})$, of a subarray, $A_i^{(j)}$, to be the absolute value of the difference between the number of 1's currently in $A_i^{(j)}$ and the number that should be in $A_i^{(j)}$ in a final sorting of A.

4.2 Establishing the Correctness of Reduce

Since the Reduce algorithm comprises the main component of the Zig-zag Sort algorithm, let us begin our detailed discussion of the correctness of Zig-zag Sort by establishing essential properties of the Reduce algorithm.

THEOREM 3. *Given an α-halver procedure, Halver, for $\alpha \leq 1/6$, which operates on arrays whose size, n, is a power of 2, then Reduce is a $(\delta, 5/6)$-attenuator, for $\delta \geq \alpha + \alpha\delta + \delta^2$.*

PROOF. W.l.o.g., let us analyze the number of 1's that end up in A; the arguments bounding the number of 0's that end up in B are similar. Let $k \leq (5/6)n$ denote the number of 1's in $A \cup B$. Also, just after the first call to Halver in Reduce, let k_1 denote the number of 1's in A and let k_2 denote the number in B, so $k = k_1 + k_2$. Moreover, because we preface our call to Attenuate in Reduce with the above-mentioned α-halver operation, $k_1 \leq \alpha k \leq (5\alpha/6)n$. Also, note that if we let k_1' denote the number of 1's in A before we perform this first α-halver operation, then $k_1 \leq k_1'$, since any α-halver operation with A as the first argument can only decrease the number of 1's in A. Note that if $n \leq 8$, then we satisfy the claimed bound, since we reduce the number of 1's in A to 0 in this case.

Suppose, inductively, that the recursive calls to Attenuate perform $(\delta, 5/6)$-attenuator operations, under the assumption that the number of 1's passed to the first recursive call in Attenuate is at most $(5/6)n/2$ and that there are at most $(5/6)n/4$ passed to the second. If $\alpha \leq 1/6$, then the results of lines 4 and 6 give us $D(A_1^{(1)}) \leq \alpha k_1$ and $D(A_2^{(1)}) \leq k_1$. Thus, inductively, after the first call to Attenuate, we have $D(A_2^{(1)}) \leq \delta k_1$. The results of lines 9 and 11 give us $D(A_1^{(2)}) \leq \alpha\delta k_1$ and $D(A_2^{(2)}) \leq \delta k_1$. Thus, inductively, after the second call to Attenuate, we have $D(A_2^{(2)}) \leq \delta^2 k_1$. Therefore, if we can show that the number of 1's passed to each call of Attenuate is 5/6 of the size of the input subarrays, then we will establish the lemma, provided that

$$\delta \geq \alpha + \alpha\delta + \delta^2.$$

To bound the number of 1's passed to each recursive call to Attenuate, we establish the following claim.

Claim: *The number of 1's passed to the first recursive call in Attenuate is at most $5n/12$.*

Since the structure of the Attenuate algorithm involves the same kinds of α-halver operations from the first recursive call to the second, this will also imply that the number of 1's passed to the second recursive call is at most $5n/24$, provided it holds for the first call. To keep the constant factors reasonable, we distinguish three cases to prove the above claim:

1. Suppose $k_2 \leq n/2$. Since $k_1 \leq \alpha k$, in this case, $k \leq n/(2-2\alpha)$, since $k = k_1 + k_2 \leq \alpha k + n/2$. Here, the number of 1's passed to the recursive call is at most $2\alpha k + \alpha n/2$, since we start with $k_1 \leq \alpha k$ and $k_2 \leq k$, and Halver$(B_1^{(1)}, B_2^{(1)})$ reduces the number of 1's in $B_1^{(1)}$ in this case to be at most $\alpha k + \alpha n/2$, by Lemma 2. Thus, since, in this case,
$$2\alpha k + \alpha n/2 \leq \alpha n/(1-\alpha) + \alpha n/2,$$
the number of 1's passed to the recursive call is at most $5n/12$ if $\alpha \leq 1/4.5$.
2. Suppose $n/2 < k_2 \leq 2n/3$. Here, the number of 0's in B is $n - k_2 < n/2$; hence, the number of 0's in $B_2^{(1)}$ is at most $\alpha(n - k_2)$, which means that the number of 1's in $B_2^{(1)}$ is at least $n/2 - \alpha(n - k_2)$, and this, in turn, implies that the number of 1's in $B_1^{(1)}$ is at most $5n/12$ if $\alpha \leq 1/6$.
3. Suppose $2n/3 < k_2 \leq 5n/6$. Of course, we also know that $k \leq 5n/6$ in this case. Here, the number of 0's in B is $n - k_2 < n/3$; hence, the number of 0's in $B_2^{(1)}$ is at most $\alpha(n - k_2)$, which means that the number of 1's in $B_2^{(1)}$ is at least $n/2 - \alpha(n - k_2)$, and this, in turn, implies that the number of 1's in $B_1^{(1)}$ is at most $k_2 - n/2 + \alpha(n - k_2)$. Thus, the number of ones in the first recursive call is at most $5n/12$ if $\alpha \leq 1/4$.

Thus, we have established the claim, which in turn, establishes that Reduce is a $(\delta, 5/6)$-attenuator, for $\delta \geq \alpha + \alpha\delta + \delta^2$, assuming $\alpha \leq 1/6$, since $k_1 \leq k_1'$. □

So, for example, using a $(1/15)$-halver as the Halver procedure implies that Reduce is a $(1/12, 5/6)$-attenuator. In addition, we have the following.

THEOREM 4. *Given an α-halver procedure, Halver, for $\alpha \leq 1/6$, which operates on arrays whose size, n, is a power of 2, then the Reduce algorithm is an $(\alpha\delta, 5/6)$-halver, for $\delta \geq \alpha + \alpha\delta + \delta^2$.*

PROOF. Let us analyze the number of 1's that may remain in the first array, A, in a call to Reduce(A, B), as the method for bounding the number of 0's in B is similar. After the first call to the Halver procedure, the number of 1's in A is at most αk, where $k \leq (5/6)n$ is the number of 1's in $A \cup B$. Then, since the Attenuate algorithm prefaced by an α-halver is a $(\delta, 5/6)$-attenuator, by Theorem 3, it will further reduce the number of 1's in A to be at most $\alpha\delta k$, where $\delta \geq \alpha + \alpha\delta + \delta^2$. Thus, this process is an $(\alpha\delta, 5/6)$-halver. □

So, for example, if we construct the Halver procedure to be a $(1/15)$-halver, then Reduce is a $(1/180, 5/6)$-halver, by Theorems 3 and 4. In addition, we have the following.

THEOREM 5. *Given an α-halver procedure, Halver, for $\alpha \leq 1/8$, which operates on arrays whose size, n, is a power of 2, then, when prefaced by an α-halver operation, the Attenuate algorithm is an ϵ-halver for $\epsilon = \alpha^2(\lceil \log(1/\alpha) \rceil + 3)$.*

PROOF. Manos [19] provides an algorithm for leveraging an α-halver to construct an ϵ-halver, for

$$\epsilon = \alpha^2(\lceil \log(1/\alpha) \rceil + 3),$$

and every call to an α-halver made in the algorithm by Manos is also made in Reduce. In addition, all the other calls to the α-halver procedure made in Reduce either keep the number of 1's in A unchanged or possibly make it even smaller, since they involve compare-exchanges between subarrays of A and B or they involve compare-exchanges done after the same ones as in Manos' algorithm (and the compare-exchanges in the Reduce algorithm never involve zig-zag swaps). Thus, the bound derived by Manos [19] for his algorithm also applies to Reduce. □

So, for example, if we take $\alpha = 1/15$, then Reduce is a $(1/32)$-halver.

4.3 The Correctness of Zig-zag Sort

The main theorem that establishes the correctness of the Zig-zag Sort algorithm, given in Figure 1, is the following.

THEOREM 6. *If it is implemented using a linear-time α-halver, Halver, for $\alpha \leq 1/15$, Zig-zag Sort correctly sorts an array of n comparable items in $O(n \log n)$ time.*

The details of the proof of Theorem 6 are given in the appendix, but let us nevertheless provide a sketch of the main ideas behind the proof here.

Recall that in each iteration, j, of Zig-zag Sort, we divide the array, A, into 2^j subarrays, $A_1^{(j)}, \ldots, A_{2^j}^{(j)}$. Applying the 0-1 principle, let us assume that A stores some number, K, of 0's and $n-K$ 1's; hence, in a final sorting of A, a subarray, $A_i^{(j)}$, should contain all 0's if $i < \lfloor K/2^j \rfloor$ and all 1's if $i > \lceil K/2^j \rceil$. Without loss of generality, let us assume $0 < K < n$, and let us define the index K to be the ***cross-over*** point in A, so that in a final sorting of A, we should have $A[K] = 0$ and $A[K+1] = 1$, by the 0-1 principle.

The overall strategy of our proof of correctness is to define a set of potential functions upper-bounding the dirtiness of the subarrays in iteration j while satisfying the following constraints:

1. The potential for any subarray, other than the one containing the cross-over point, should be less than its size, with the potential of any subarray being a function of its distance from the cross-over point.
2. The potential for any subarray should be reduced in an iteration of Zig-zag Sort by an amount sufficient for its two "children" subarrays to satisfy their dirtiness potentials for the next iteration.
3. The total potential of all subarrays that should contain only 0's (respectively, 1's) should be bounded by the size of a single subarray.

The first constraint ensures that A will be sorted when we are done, since the size of each subarray at the end is 1. The second constraint is needed in order to maintain bounds on the potential functions from one iteration to the next. And the third constraint is needed in order to argue that the dirtiness in A is concentrated around the cross-over point.

So as to prepare for defining the particular set of ***dirtiness invariants*** for iteration j that we will show inductively holds after the splitting step in each iteration j, let us introduce a few additional definitions. Define the ***uncertainty interval*** to be the set of indices for cells in A with indices in the interval,

$$[K - n_j/2,\ K + 1 + n_j/2],$$

where $n_j = n/2^j$ is the size of each subarray, $A_i^{(j)}$, and K is the cross-over point in A. Note that this implies that there are exactly two subarrays that intersect the uncertainty interval in iteration j of Zig-zag Sort. In addition, for any subarray, $A_i^{(j)}$, define $d_{i,j}$ to be the number of iterations since this subarray has had an ancestor that was intersecting the uncertainty interval for that level. Also, let m_0 denote the smallest index, i, such that $A_i^{(j)}$ has a cell in the uncertainty interval and let m_1 denote the largest index, i, such that $A_i^{(j)}$ has a cell in the uncertainty interval (we omit an implied dependence on j here). Note that these indices are defined for the sake of simplifying our notation, since $m_1 = m_0 + 1$.

Then, given that the Reduce algorithm is simultaneously an ϵ-halver, a $(\beta, 5/6)$-halver, and a $(\delta, 5/6)$-attenuator, with the parameters, ϵ, β, and δ, depending on α, the parameter for the α-halver, Halver, as discussed in the previous section, our potential functions and dirtiness invariants are as follows:

1. After the splitting step, for any subarray, $A_i^{(j)}$, for $i \leq m_0 - 1$ or $i \geq m_1 + 1$,
$$D(A_i^{(j)}) \leq 4^{d_{i,j}} \delta^{d_{i,j}-1} \beta n_j.$$
2. If the cross-over point, K, indexes a cell in $A_{m_0}^{(j)}$, then $D(A_{m_1}^{(j)}) \leq n_j/6$.
3. If the cross-over point, K, indexes a cell in $A_{m_1}^{(j)}$, then $D(A_{m_0}^{(j)}) \leq n_j/6$.

Our proof of correctness, then, is based on arguing how the outer-zig and outer-zag phases of Zig-zag Sort reduce the dirtiness of each subarray sufficiently to allow the dirtiness invariants to hold for the next iteration.

5. CONSTANT FACTORS

An essential building block for Zig-zag Sort is the existence of α-halvers for moderately small constant values of α, with $\alpha \leq 1/15$ being sufficient for correctness, based on the analysis. Suppose such an algorithm uses cn compare-exchange operations, for two subarrays whose combined size is n. Then the number of compare-exchange operations performed by the Attenuate algorithm is characterized by the recurrence,

$$T(n) = T(n/2) + T(n/4) + 2.25cn,$$

where n is the total combined size of the arrays, A and B; hence, $T(n) = 9cn$. Thus, the running time, in terms of compare-exchange operations, for Reduce, is $10cn$, which implies that the running time for Zig-zag Sort, in terms of compare-exchange operations, is at most $50cn \log n$.

An algorithm for performing a data-oblivious α-halver operation running in $O(n)$ time can be built from constructions for bipartite (γ, t)-expander graphs (e.g., see [1,2,14,29]), where such a graph, $G = (X, Y, E)$, has the property that any subset $S \subset X$ of size at most $t|X|$ has at least $\gamma|S|$ neighbors in Y, and similarly for going from Y to X. Thus, if set $A = X$ and $B = Y$ and we use the edges for compare-exchange operations, then we can build an α-halver from a $((1-\alpha)/\alpha, \alpha)$-expander graph with $|X| = |Y| = n$. Also, notice that such a bipartite graph, G', can be constructed from a non-bipartite expander graph, G, on n vertices, simply by making two copies, v' and v'', in G', for every vertex v in G, and replacing each edge (v, w) in G with the edge (v', w''). Note, in addition, that G and G' have the same number of edges.

The original AKS sorting network [1, 2] is based on the use of ϵ-halvers for very small constant values of ϵ and is estimated to have a depth of roughly $2^{100} \log n$, meaning that the running time for simulating it sequentially would run in roughly $2^{99} n \log n$ time in terms of compare-exchange operations (since implementing an ϵ-halver sequentially halves the constant factor in the depth, given that every compare-exchange is between two items). Seiferas [27] describes an improved scheme for building a variant of the AKS sorting network to have $6.05 \log n$ iterations, each seven (1/402.15)-halvers deep.

By several known results (e.g., see [14,29]), one can construct an expander graph, as above, which can be used as an ϵ-halver, using a k-regular graph with

$$k = \frac{2(1-\epsilon)(1-\epsilon+\sqrt{1-2\epsilon})}{\epsilon^2}.$$

So, for example, if $\epsilon = 1/15$, then we can construct an ϵ-halver with cn edges, where $c = 392$. Using this construction results

in a running time for Zig-zag Sort of $19\,600n \log n$, in terms of compare-exchange operations. For the network of Seiferas [27], on the other hand, using such an ϵ-halver construction, one can design such a (1/402.15)-halver to have degree $k = 642\,883$; hence, the running time of the resulting sorting algorithm would have an upper bound of $13\,613\,047n \log n$, in terms of compare-exchange operations.

There are also non-constructive results for proving the existence of ϵ-halvers and sorting networks. Paterson [20] shows that k-regular ϵ-halvers exist with

$$k = \lceil (2 \log \epsilon)/\log(1-\epsilon) + 2/\epsilon - 1 \rceil.$$

So, for example, there is a $(1/15)$-halver with $54n$ edges, which would imply a running time of $2700n \log n$ for Zig-zag Sort. Using the above existence bound for the (1/402.15)-halvers used in the network of Seiferas [27], on the other hand, results in a running time of $119\,025n \log n$. Alternatively, Paterson [20] shows that there exists a sorting network of depth roughly $6100 \log n$ and Chvátal [6] shows that there exists a sorting network of depth $1830 \log n$, for $n \geq 2^{78}$. Therefore, a non-constructive version of Zig-zag Sort is competitive with these non-constructive versions of the AKS sorting network, while also being simpler.

Acknowledgments

This research was supported in part by the National Science Foundation under grants 1011840, 1217322, 0916181 and 1228639, and by the Office of Naval Research under MURI grant N00014-08-1-1015. We would like to thank Daniel Hirschberg for several helpful comments regarding an earlier version of this paper.

6. REFERENCES

[1] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *15th ACM Symp. on Theory of Computing (STOC)*, pages 1–9. ACM, 1983.

[2] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3(1):1–19, 1983.

[3] M. Ajtai, J. Komlós, and E. Szemerédi. Halvers and expanders. In *33rd IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 686–692, 1992.

[4] S. W. Al-Haj Baddar and K. E. Batcher. *Designing Sorting Networks*. Springer, 2011.

[5] K. E. Batcher. Sorting networks and their applications. In *Proc. of the Spring Joint Computer Conference (AFIPS)*, pages 307–314. ACM, 1968.

[6] V. Chvátal. Lecture notes on the new AKS sorting network. Technical report, Rutgers Univ., 1992. ftp://ftp.cs.rutgers.edu/pub/technical-reports/dcs-tr-294.ps.Z.

[7] R. Cypher. A lower bound on the size of Shellsort sorting networks. *SIAM Journal on Computing*, 22(1):62–71, 1993.

[8] I. Damgøard, S. Meldgaard, and J. Nielsen. Perfectly secure oblivious RAM without random oracles. In Y. Ishai, editor, *Theory of Cryptography (CRYPTO)*, volume 6597 of *LNCS*, pages 144–163. Springer, 2011.

[9] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, May 1996.

[10] M. T. Goodrich. Randomized shellsort: A simple data-oblivious sorting algorithm. *J. ACM*, 58(6):27:1–27:26, Dec. 2011.

[11] M. T. Goodrich. Spin-the-bottle sort and annealing sort: Oblivious sorting via round-robin random comparisons. *Algorithmica*, pages 1–24, 2012.

[12] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In L. Aceto, M. Henzinger, and J. Sgall, editors, *Int. Conf. on Automata, Languages and Programming (ICALP)*, volume 6756 of *LNCS*, pages 576–587. Springer, 2011.

[13] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *23rd Symp. on Discrete Algorithms (SODA)*, pages 157–167, 2012.

[14] S. Hoory, N. Linial, and A. Wigderson. Expander graphs and their applications. *Bull. Amer. Math. Soc.*, 43:439–561, 2006.

[15] J. Incerpi and R. Sedgewick. Improved upper bounds on Shellsort. *Journal of Computer and System Sciences*, 31(2):210 – 224, 1985.

[16] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.

[17] T. Leighton. Tight bounds on the complexity of parallel sorting. In *16th ACM Symp. on Theory of Computing (STOC)*, pages 71–80. ACM, 1984.

[18] T. Leighton and C. Plaxton. Hypercubic sorting networks. *SIAM Journal on Computing*, 27(1):1–47, 1998.

[19] H. Manos. Construction of halvers. *Information Processing Letters*, 69(6):303–307, 1999.

[20] M. S. Paterson. Improved sorting networks with $O(\log N)$ depth. *Algorithmica*, 5(1-4):75–92, 1990.

[21] N. Pippenger. Communication networks. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science (vol. A)*, pages 805–833. MIT Press, 1990.

[22] C. Plaxton, B. Poonen, and T. Suel. Improved lower bounds for Shellsort. In *33rd Symp. on Foundations of Computer Science (FOCS)*, pages 226–235, 1992.

[23] C. Plaxton and T. Suel. Lower bounds for Shellsort. *Journal of Algorithms*, 23(2):221–240, 1997.

[24] B. Poonen. The worst case in Shellsort and related algorithms. *Journal of Algorithms*, 15(1):101–124, 1993.

[25] V. R. Pratt. *Shellsort and sorting networks*. PhD thesis, Stanford University, Stanford, CA, USA, 1972.

[26] R. Sedgewick. Analysis of Shellsort and related algorithms. In J. Diaz and M. Serna, editors, *European Symp. on Algorithms (ESA)*, volume 1136 of *LNCS*, pages 1–11. Springer, 1996.

[27] J. Seiferas. Sorting networks of logarithmic depth, further simplified. *Algorithmica*, 53(3):374–384, 2009.

[28] D. L. Shell. A high-speed sorting procedure. *Comm. ACM*, 2(7):30–32, July 1959.

[29] H. Xie. Studies on sorting networks and expanders, 1998. Thesis, Ohio University, https://etd.ohiolink.edu/.

APPENDIX

As outlined above, our proof of Theorem 6, establishing the correctness of Zig-zag Sort, is based on our characterizing the dirtiness invariant for the subarrays in A, from one iteration of Zig-zag Sort to the next. Let us therefore assume we have satisfied the dirtiness invariants for a given iteration and let us now consider how the compare-exchange operations in a given iteration impact the dirtiness bounds for each subarray. We establish such bounds by considering how the Reduce algorithm impacts various subarrays in the outer-zig and outer-zag steps, according to the order in which Reduce is called and the distance of the different subarrays from the cross-over point.

Recall the potential functions for our dirtiness invariants, with

$n_j = n/2^j$. One immediate consequence of these bounds is the following.

LEMMA 7 (CONCENTRATION OF DIRTINESS LEMMA). *The total dirtiness of all the subarrays from the subarray, $A_1^{(j)}$, to the subarray $A_{m_0-1}^{(j)}$, or from the subarray, $A_{m_1+1}^{(j)}$, to the subarray $A_{2^j}^{(j)}$, after the splitting step, is at most*

$$\frac{8\beta n_j}{1-8\delta},$$

provided $\delta < 1/8$.

PROOF. Note that, since we divide each subarray in two in each iteration of Zig-zag Sort, there are 2^k subarrays, $A_i^{(j)}$, with depth $k = d_{i,j}$. Thus, by the dirtiness invariants, the total dirtiness of all the subarrays from the first subarray, $A_1^{(j)}$, to the subarray $A_{i+1}^{(j)}$, after the splitting step, for $j < m_0$, is at most

$$\begin{aligned}\sum_{k=1}^{j} 2^k 4^k \delta^{k-1}\beta n_j &< 8\beta n_j \sum_{k=0}^{\infty}(8\delta)^k \\ &= \frac{8\beta n_j}{1-8\delta},\end{aligned}$$

provided $\delta < 1/8$. A similar argument establishes the bound for the total dirtiness from the subarray, $A_{m_1+1}^{(j)}$, to the subarray $A_{2^j}^{(j)}$. □

For any subarray, B, of A, define $\overrightarrow{D}(B)$ to be the dirtiness of B after the outer-zig step and $\overleftarrow{D}(B)$ to be the dirtiness of B after the outer-zag step. Using this notation, we begin our analysis with the following lemma, which establishes a dirtiness bound for subarrays far to the left of the cross-over point.

LEMMA 8 (LOW-SIDE ZIG LEMMA). *Suppose the dirtiness invariants are satisfied after the splitting step in iteration j. Then, for $i \leq m_0 - 2$, after the first (outer zig) phase in iteration j,*

$$\overrightarrow{D}(A_i^{(j)}) \leq \delta D(A_{i+1}^{(j)}) \leq 4^{d_{i+1,j}}\delta^{d_{i+1,j}}\beta n_j,$$

provided $\delta \leq 1/12$ and $\beta \leq 1/180$.

PROOF. Suppose $i \leq m_0 - 2$. Assuming that the dirtiness invariants are satisfied after the splitting step in iteration j, then, prior to the swaps done in the inner zig-zag step for the subarrays $A_i^{(j)}$ and $A_{i+1}^{(j)}$, we have

$$D(A_{i+1}^{(j)}) \leq 4^{d_{i+1,j}}\delta^{d_{i+1,j}-1}\beta n_j.$$

In addition, by Lemma 7, the total dirtiness of all the subarrays from the first subarray, $A_1^{(j)}$, to the subarray $A_i^{(j)}$, after the splitting step, is at most

$$\frac{8\beta n_j}{1-8\delta} \leq \frac{n_j}{6},$$

provided $\delta \leq 1/12$ and $\beta \leq 1/180$. Moreover, the cumulative way that we process the subarrays from the first subarray to $A_i^{(j)}$ implies that the total amount of dirtiness brought rightward from these subarrays to $A_i^{(j)}$ is at most the above value. Therefore, after the swap of $A_i^{(j)}$ and $A_{i+1}^{(j)}$ in the inner zig-zag step, the $(\delta, 5/6)$-attenuator, Reduce, will be effective to reduce the dirtiness for $A_i^{(j)}$ so that

$$\overrightarrow{D}(A_i^{(j)}) \leq \delta D(A_{i+1}^{(j)}) \leq 4^{d_{i+1,j}}\delta^{d_{i+1,j}}\beta n_j.$$

□

As discussed at a high level earlier, the above proof provides a motivation for the inner zig-zag step, which might at first seem counter-intuitive, since it swaps many pairs of items that are likely to be in the correct order already. The reason we perform the inner zig-zag step, though, is that, as we reasoned in the above proof, it provides a way to "shovel" relatively large amounts of dirtiness, while reducing the dirtiness of all the subarrays along the way, starting with the first subarray in A. In addition to the above Low-side Zig Lemma, we have the following for a subarray close to the uncertainty interval.

LEMMA 9 (LEFT-NEIGHBOR ZIG LEMMA). *Suppose the dirtiness invariants are satisfied after the splitting step in iteration j. If $i = m_0 - 1$, then, after the first (outer zig) phase in iteration j,*

$$\overrightarrow{D}(A_i^{(j)}) \leq \beta n_j,$$

provided $\delta \leq 1/12$, $\epsilon \leq 1/32$, and $\beta \leq 1/180$.

PROOF. Since $i + 1 = m_0$, we are considering in this lemma impact of calling Reduce on $A_i^{(j)}$ and $A_{i+1}^{(j)} = A_{m_0}^{(j)}$, that is, $A_i^{(j)}$ and the left subarray that intersects the uncertainty interval. There are two cases.

Case 1: The cross-over point, K, is in $A_{m_1}^{(j)}$. By the dirtiness invariant, and Lemma 7, even if this outer zig step has brought all the 1's from the left rightward, the total number of 1's in $A_i^{(j)} \cup A_{m_0}^{(j)}$ at this point is at most

$$\frac{n_j}{6} + \frac{8\beta n_j}{1-8\delta} \leq \frac{n_j}{3},$$

provided $\delta \leq 1/12$ and $\beta \leq 1/180$. The above dirtiness is clearly less than $5n_j/6$ in this case. Thus, the Reduce algorithm, which is an $(\beta, 5/6)$-halver, is effective in this case to give us

$$\overrightarrow{D}(A_i^{(j)}) \leq \beta n_j.$$

Case 2: The cross-over point, K, is in $A_{m_0}^{(j)}$. Suppose we were to sort the items currently in $A_i^{(j)} \cup A_{m_0}^{(j)}$. Then, restricted to the current state of these two subarrays, we would get current cross-over point, K', which could be to the left, right, or possibly equal to the real one, K. Note that each 0 that is currently to the right of $A_{m_0}^{(j)}$ implies there must be a corresponding 1 currently placed somewhere from $A_1^{(j)}$ to $A_{m_0}^{(j)}$, possibly even in $A_i^{(j)} \cup A_{m_0}^{(j)}$. By the dirtiness invariants for iteration j, the number of such additional 1's is bounded by

$$\frac{n_j}{6} + \frac{8\beta n_j}{1-8\delta} \leq \frac{n_j}{3},$$

provided $\delta \leq 1/12$ and $\beta \leq 1/180$. Thus, since there the number of 1's in $A_{m_0}^{(j)}$ is supposed to be at most $n_j/2$, based on the location of the cross-over point for this case (if the cross-over was closer than $n_j/2$ to the ith subarray, then i would be m_0), the total number of 1's currently in $A_i^{(j)} \cup A_{m_0}^{(j)}$ is at most $n_j/3 + n_j/2 = 5n_j/6$. Therefore, the Reduce algorithm, which is a $(\beta, 5/6)$-halver, is effective in this case to give us

$$\overrightarrow{D}(A_i^{(j)}) \leq \beta n_j.$$

□

There is also the following.

LEMMA 10 (STRADDLING ZIG LEMMA). *Suppose the dirtiness invariants are satisfied after the splitting step in iteration j. Provided $\delta \leq 1/12$, $\epsilon \leq 1/32$, and $\beta \leq 1/180$, then after the step in first (outer zig) phase in iteration j, comparing $A_{m_0}^{(j)}$ and $A_{m_1}^{(j)}$, we have the following:*

1. If K is in $A^{(j)}_{m_0}$, then

$$\overrightarrow{D}(A^{(j)}_{m_1}) \leq n_j/6 - \beta n_j.$$

2. If K is in $A^{(j)}_{m_1}$, then

$$\overrightarrow{D}(A^{(j)}_{m_0}) \leq n_j/6.$$

PROOF. Please see http://arxiv.org/abs/1403.2777 for details. □

In addition, we have the following.

LEMMA 11 (RIGHT-NEIGHBOR ZIG LEMMA). *Suppose that the dirtiness invariant is satisfied after the splitting step in iteration j. If $i = m_1 + 1$, then, after the step comparing subarray $A^{(j)}_{m_1}$ and subarray $A^{(j)}_i$ in the first (outer zig) phase in iteration j,*

$$\overrightarrow{D}(A^{(j)}_i) \leq \beta n_j,$$

provided $\delta \leq 1/12$, $\epsilon \leq 1/32$, and $\beta \leq 1/180$. Also, if the cross-over is in $A^{(j)}_{m_0}$, then $\overrightarrow{D}(A^{(j)}_{m_1}) \leq n_j/6$.

PROOF. Since $i - 1 = m_1$, we are considering in this lemma the result of calling Reduce on $A^{(j)}_{i-1} = A^{(j)}_{m_1}$ and $A^{(j)}_i$. There are two cases.

Case 1: The cross-over point, K, is in $A^{(j)}_{m_0}$. In this case, by the dirtiness invariant for $A^{(j)}_i$ and the previous lemma, the total number of 0's in $A^{(j)}_i \cup A^{(j)}_{m_1}$ at this point is at most $n_j/6$; hence, $\overrightarrow{D}(A^{(j)}_{m_1}) \leq n_j/6$ after this comparison. In addition, in this case, the $(\beta, 5/6)$-halver algorithm, Reduce, is effective to give us

$$\overrightarrow{D}(A^{(j)}_i) \leq \beta n_j.$$

Case 2: The cross-over point, K, is in $A^{(j)}_{m_1}$. Suppose we were to sort the items currently in $A^{(j)}_{m_1} \cup A^{(j)}_i$. Then, restricted to these two subarrays, we would get a cross-over point, K', which could be to the left, right, or possibly equal to the real one, K. Note that each 1 that is currently to the left of $A^{(j)}_{m_1}$ implies there must be a corresponding 0 currently placed somewhere from $A^{(j)}_{m_1}$ to $A^{(j)}_{2^j}$, possibly even in $A^{(j)}_{m_1} \cup A^{(j)}_i$. The bad scenario with respect to dirtiness for $A^{(j)}_i$ is when 0's are moved into $A^{(j)}_{m_1} \cup A^{(j)}_i$.

By Lemmas 8, 9, and 10, the number of such additional 0's is bounded by

$$\frac{\beta n_j}{1 - 8\delta} + \frac{n_j}{6}.$$

Thus, since the number of 0's in $A^{(j)}_{m_1}$, based on the location of the cross-over point, is supposed to be at most $n_j/2$, the total number of 0's currently in $A^{(j)}_i \cup A^{(j)}_{m_1}$ is at most

$$\frac{\beta n_j}{1 - 8\delta} + \frac{2n_j}{3} \leq 5n_j/6,$$

provided $\delta \leq 1/12$, $\epsilon \leq 1/32$, and $\beta \leq 1/180$. Therefore, the $(\beta, 5/6)$-halver algorithm, Reduce, is effective in this case to give us

$$\overrightarrow{D}(A^{(j)}_i) \leq \beta n_j.$$

□

Note that the above lemma covers the case just before we do the next inner zig-zag step involving the subarrays $A^{(j)}_i$ and $A^{(j)}_{i+1}$. For bounding the dirtiness after this inner zig-zag step we have the following.

LEMMA 12 (HIGH-SIDE ZIG LEMMA). *Suppose the dirtiness invariant is satisfied after the splitting step in iteration j. Provided $\delta \leq 1/12$, $\epsilon \leq 1/32$, and $\beta \leq 1/180$, then, for $m_1 + 1 \leq i < 2^j$, after the first (outer zig) phase in iteration j,*

$$\overrightarrow{D}(A^{(j)}_i) \leq D(A^{(j)}_{i+1}) + \delta^{i-m_1-1}\beta n_j.$$

PROOF. The proof is by induction, using Lemma 11 as the base case. We assume inductively that before we do the swaps for the inner zig-zag step, $D(A^{(j)}_i) \leq \delta^{i-m_1-1}\beta n_j$. So after we do the swapping for the inner zig-zag step and an $(\delta, 5/6)$-attenuator algorithm, Reduce, we have $\overrightarrow{D}(A^{(j)}_i) \leq D(A^{(j)}_{i+1}) + \delta^{i-m_1-1}\beta n_j$ and $\overrightarrow{D}(A^{(j)}_{i+1}) \leq \delta^{i-m_1}\beta n_j$. □

In addition, by the induction from Lemma 12's proof, $\overrightarrow{D}(A^{(j)}_{2^j}) \leq \delta^{2^j-m_1-1}\beta n_j$, after we complete the outer zig phase. So let us consider the changes caused by the outer zag phase.

LEMMA 13 (HIGH-SIDE ZAG LEMMA). *Suppose the dirtiness invariant is satisfied after the splitting step in iteration j. Then, for $m_1 + 2 \leq i \leq 2^j$, after the second (outer zag) phase in iteration j,*

$$\begin{aligned}\overleftarrow{D}(A^{(j)}_i) \leq \delta\overrightarrow{D}(A^{(j)}_{i-1}) &\leq \delta D(A^{(j)}_i) + \delta^{i-m_1-1}\beta n_j \\ &\leq 4^{d_{i,j}}\delta^{d_{i,j}}\beta n_j + \delta^{i-m_1-1}\beta n_j,\end{aligned}$$

provided $\delta \leq 1/12$, $\epsilon \leq 1/32$, and $\beta \leq 1/180$.

PROOF. By Lemmas 7 and 12, and a simple induction argument, just before we do the swaps for the inner zig-zag step,

$$\overrightarrow{D}(A^{(j)}_{i-1}) \leq D(A^{(j)}_i) + \delta^{i-m_1-2}\beta n_j$$

and

$$\overrightarrow{D}(A^{(j)}_i) \leq \frac{8\beta n_j}{1 - 8\delta}.$$

We then do the inner zig-zag swaps and, provided $\delta \leq 1/12$, $\epsilon \leq 1/32$, and $\beta \leq 1/180$, we have a small enough dirtiness to apply the $(\delta, 5/6)$-attenuator, Reduce, effectively, which completes the proof. □

In addition, we have the following.

LEMMA 14 (RIGHT-NEIGHBOR ZAG LEMMA). *Suppose the dirtiness invariant is satisfied after the splitting step in iteration j. If $i = m_1 + 1$, then, after the second (outer zag) phase in iteration j,*

$$\overleftarrow{D}(A^{(j)}_i) \leq \beta n_j.$$

provided $\delta \leq 1/12$, $\epsilon \leq 1/32$, and $\beta \leq 1/180$.

PROOF. Since $i - 1 = m_1$, we are considering in this lemma the result of calling Reduce on $A^{(j)}_i$ and $A^{(j)}_{i-1} = A^{(j)}_{m_1}$. There are two cases.

Case 1: The cross-over point, K, is in $A^{(j)}_{m_0}$. In this case, by the previous lemmas, bounding the number of 0's that could have come to this place from previously being in or to the right of $A^{(j)}_{m_1}$, the total number of 0's in $A^{(j)}_i \cup A^{(j)}_{m_1}$ at this point is at most

$$\frac{8\beta n_j}{1 - 8\delta} + n_j/6 \quad < \quad \frac{5n_j}{6},$$

provided $\delta \leq 1/12$, $\epsilon \leq 1/32$, and $\beta \leq 1/180$. Thus, the $(\beta, 5/6)$-halver algorithm, Reduce, is effective in this case to give us

$$\overleftarrow{D}(A^{(j)}_i) \leq \beta n_j.$$

Case 2: The cross-over point, K, is in $A^{(j)}_{m_1}$. Suppose we were to sort the items currently in $A^{(j)}_i \cup A^{(j)}_{m_1}$. Then, restricted to these two subarrays, we would get a cross-over point, K', which could be to the left, right, or possibly equal to the real one, c. Note that each 1 that is currently in or to the left of $A^{(j)}_{m_0}$ implies there must be a corresponding 0 currently placed somewhere from $A^{(j)}_{m_1}$ to $A^{(j)}_{2^j}$, possibly even in $A^{(j)}_{m_1} \cup A^{(j)}_i$. The bad scenario with respect to dirtiness for $A^{(j)}_i$ is when 0's are moved into $A^{(j)}_i \cup A^{(j)}_{m_1}$. By the previous lemmas, the number of such additional 0's (that is, 1's currently in or to the left of $A^{(j)}_{m_0}$) is bounded by

$$\frac{\beta n_j}{1-8\delta} + n_j/6.$$

Thus, since the number of 0's that are supposed to be in $A^{(j)}_{m_1}$ is at most $n_j/2$ based on the location of the cross-over point, the total number of 0's currently in $A^{(j)}_i \cup A^{(j)}_{m_1}$ is at most

$$\frac{\beta n_j}{1-8\delta} + n_j/6 + n_j/2 \leq 5n_j/6,$$

provided $\delta \leq 1/12$, $\epsilon \leq 1/32$, and $\beta \leq 1/180$. Therefore, the $(\beta, 5/6)$-halver algorithm, Reduce, is effective in this case to give us

$$\overleftarrow{D}(A^{(j)}_i) \leq \beta n_j.$$

□

Next, we have the following.

LEMMA 15 (STRADDLING ZAG LEMMA). *Suppose the dirtiness invariant is satisfied after the splitting step in iteration j and $\delta \leq 1/12$, $\epsilon \leq 1/32$, and $\beta \leq 1/180$. Then, after the comparison of $A^{(j)}_{m_0}$ and $A^{(j)}_{m_1}$,*

1. *If $K + n_j/4$ indexes a cell in $A^{(j)}_{m_0}$, then after the second (outer zag) phase, $\overleftarrow{D}(A^{(j)}_{m_1}) \leq \beta n_j$.*
2. *If $K - n_j/4$ indexes a cell in $A^{(j)}_{m_1}$, then after the second (outer zag) phase, $\overleftarrow{D}(A^{(j)}_{m_0}) \leq \beta n_j$.*
3. *Else, if K is in $A^{(j)}_{m_1}$, then $\overleftarrow{D}(A^{(j)}_{m_0}) \leq n_j/12 - \beta n_j$, and if K is in $A^{(j)}_{m_0}$, then $\overleftarrow{D}(A^{(j)}_{m_1}) \leq n_j/12$.*

PROOF. Please see http://arxiv.org/abs/1403.2777 for details. □

Next, we have the following.

LEMMA 16 (LEFT-NEIGHBOR ZAG LEMMA). *Suppose the dirtiness invariant is satisfied after the splitting step in iteration j. Then, after the call to the $(\beta, 5/6)$-halver, Reduce, comparing $A^{(j)}_i$, for $i = m_0 - 1$, and $A^{(j)}_{m_0}$ in the second (outer zag) phase in iteration j,*

$$\overleftarrow{D}(A^{(j)}_i) \leq \beta n_j,$$

provided $\delta \leq 1/12$, $\epsilon \leq 1/32$, and $\beta \leq 1/180$. Also, if the cross-over point, K, is in $A^{(j)}_{m_1}$, then $\overleftarrow{D}(A^{(j)}_{m_0}) \leq 2\beta n_j$, if $K - n_j/4$ indexes a cell in $A^{(j)}_{m_1}$, and $\overleftarrow{D}(A^{(j)}_{m_0}) \leq n_j/12$, if $K - n_j/4$ indexes a cell in $A^{(j)}_{m_0}$.

PROOF. Please see http://arxiv.org/abs/1403.2777 for details. □

Finally, we have the following.

LEMMA 17 (LOW-SIDE ZAG LEMMA). *Suppose the dirtiness invariant is satisfied after the splitting step in iteration j. If $\delta \leq 1/12$, $\epsilon \leq 1/32$, and $\beta \leq 1/180$, then, for $i \leq m_0 - 1$, after the second (outer zag) phase in iteration j,*

$$\overleftarrow{D}(A^{(j)}_i) \leq \delta D(A^{(j)}_i) + \delta^{m_0 - i - 1}\beta n_j.$$

PROOF. The proof is by induction on $m_0 - i$, starting with Lemma 16 as the basis of the induction. Before doing the swapping for the inner zig-zag step for subarray i, by Lemma 8,

$$\overrightarrow{D}(A^{(j)}_{i-1}) \leq \delta D(A^{(j)}_i)$$

and

$$\overleftarrow{D}(A^{(j)}_i) \leq \delta^{m_0 - i - 1}\beta n_j.$$

Thus, after the swaps for the inner zig-zag and the $(\delta, 5/6)$-attenuator algorithm, Reduce,

$$\overleftarrow{D}(A^{(j)}_i) \leq \delta D(A^{(j)}_i) + \delta^{m_0 - i - 1}\beta n_j.$$

and

$$\overleftarrow{D}(A^{(j)}_{i-1}) \leq \delta^{m_0 - i}\beta n_j.$$

□

This completes all the lemmas we need in order to calculate bounds for ϵ and δ that will allow us to satisfy the dirtiness invariant for iteration $j+1$ if it is satisfied for iteration j.

LEMMA 18. *Provided $\delta \leq 1/12$, $\epsilon \leq 1/32$, and $\beta \leq 1/180$, if the dirtiness invariant for iteration j is satisfied after the splitting step for iteration j, then the dirtiness invariant for iteration j is satisfied after the splitting step for iteration $j+1$.*

PROOF. Let us consider each subarray, $A^{(j)}_i$, and its two children, $A^{(j+1)}_{2i-1}$ and $A^{(j+1)}_{2i}$, at the point in the algorithm when we perform the splitting step. Let m'_0 denote the index of the lowest-indexed subarray on level $j+1$ that intersects the uncertainty interval, and let m'_1 $(= m'_0 + 1)$ denote the index of the highest-indexed subarray on level $j+1$ that intersects the uncertainty interval. Note that we either have m'_0 and m'_1 both being children of m_0, m'_0 and m'_1 both being children of m_1, or m'_0 is a child of m_0 and m'_1 is a child of m_1. That is, $m'_0 = 2m_0 - 1$, $m'_0 = 2m_0$, or $m'_0 = 2m_1 - 1 = 2m_0 + 1$, and $m'_1 = 2m_0 = 2m_1 - 2$, $m'_1 = 2m_1 - 1$, or $m'_1 = 2m_1$,

1. $i \leq m_0 - 1$. In the worst case, based on the three possibilities for m'_0 and m'_1, we need

   $$D(A^{(j+1)}_{2i-1}) \leq 4^{d_{2i-1,j+1}}\delta^{d_{2i-1,j+1}-1}\beta n_j = 4^{d_{i,j}+1}\delta^{d_{i,j}}\beta n_j$$

   and

   $$D(A^{(j+1)}_{2i}) \leq 4^{d_{2i,j+1}}\delta^{d_{2i,j+1}-1}\beta n_j = 4^{d_{i,j}+1}\delta^{d_{i,j}}\beta n_j.$$

   By Lemma 17, just before the splitting step for $A^{(j)}_i$, we have

   $$\overleftarrow{D}(A^{(j)}_i) \leq \delta D(A^{(j)}_i) + \delta^{m_0 - i - 1}\beta n_j \leq (4^{d_{i,j}}+1)\delta^{d_{i,j}}\beta n_j,$$

   and we then partition $A^{(j)}_i$, so that $n_{j+1} = n_j/2$. Thus, for either $k = 2i - 1$ or $k = 2i$, we have $D(A^{(j+1)}_k) \leq 4^{d_{i,j}+1}\delta^{d_{i,j}}\beta n_{j+1}$, which satisfies the dirtiness invariant for the next iteration.
2. $i \geq m_1 + 1$. In the worst case, based on the three possibilities for m'_0 and m'_1, we need

   $$D(A^{(j+1)}_{2i-1}) \leq 4^{d_{2i-1,j+1}}\delta^{d_{2i-1,j+1}-1}\beta n_j = 4^{d_{i,j}+1}\delta^{d_{i,j}}\beta n_j$$

and

$$D(A_{2i}^{(j+1)}) \leq 4^{d_{2i,j+1}}\delta^{d_{2i,j+1}-1}\beta n_j = 4^{d_{i,j}+1}\delta^{d_{i,j}}\beta n_j.$$

By Lemma 13, just before the splitting step for $A_i^{(j)}$, we have

$$\overleftarrow{D}(A_i^{(j)}) \leq \delta D(A_i^{(j)}) + \delta^{i-m_1-1}\beta n_j \leq (4^{d_{i,j}}+1)\delta^{d_{i,j}}\beta n_j,$$

and we then partition $A_i^{(j)}$, so that $n_{j+1} = n_j/2$. Thus, for either $k = 2i-1$ or $k = 2i$, we have $D(A_k^{(j+1)}) \leq 4^{d_{i,j}+1}\delta^{d_{i,j}}\beta n_{j+1}$, which satisfies the dirtiness invariant for the next iteration.

3. $i = m_0$. In this case, there are subcases.
   (a) Suppose $K + n_j/4$ indexes a cell in $A_{m_0}^{(j)}$. Then, by Lemma 15, $\overleftarrow{D}(A_{m_1}^{(j)}) \leq \beta n_j$. In this case, m_0' and m_1' are both children of m_0 and we need $A_{2i-1}^{(j+1)}$ to have dirtiness at most $n_{j+1}/6 = n_j/12$. The number of 1's in $A_i^{(j)}$ is bounded by $n_j/2$ (or otherwise, $i = m_1$) plus the number of additional 1's that may be here because of 0's that remain to the right, which is bounded by

$$n' = n_j/2 + \beta n_j + \frac{\beta n_j}{1-8\delta} + \frac{\delta\beta n_j}{1-\delta},$$

provided $\delta \leq 1/12$, $\epsilon \leq 1/32$, and $\beta \leq 1/180$. If $n' \leq n_j/2$, then an ϵ-halver operation applied after the split, with $\epsilon \leq 1/6$, will satisfy the dirtiness invariants for m_0' and m_1'. If, on the other hand, $n' > n_j/2$, then an ϵ-halver operation applied after the split will give us

$$\begin{aligned} D(A_{2i-1}^{(j+1)}) &\leq \epsilon n_{j+1} + (1-\epsilon)\cdot(n' - n_{j+1}) \\ &\leq n_{j+1}/6, \end{aligned}$$

provided $\delta \leq 1/12$, $\epsilon \leq 1/32$, and $\beta \leq 1/180$.

   (b) Suppose $K - n_j/4$ indexes a cell in $A_{m_1}^{(j)}$. Then, by Lemma 16, $\overleftarrow{D}(A_{m_0}^{(j)}) \leq 2\beta n_j$. In this case, we need $D(A_{2i}^{(j+1)}) \leq 4\beta n_{j+1}$ and $D(A_{2i-1}^{(j+1)}) \leq 4\beta n_{j+1}$, both of which which follow from the above bound.
   (c) Suppose neither of the previous subcases hold. Then we have two possibilities:
      i. Suppose K indexes a cell in $A_{m_1}^{(j)}$. Then $\overleftarrow{D}(A_{m_0}^{(j)}) \leq n_j/12$, by Lemma 16. We need $D(A_{2i}^{(j+1)}) \leq n_{j+1}/6$, which follows immediately from this bound, and we also need $D(A_{2i-1}^{(j+1)}) \leq 4\beta n_{j+1}$, which follows by our performing a Reduce step, which is a $(\beta, 5/6)$-halver, after we do our split.
      ii. Suppose K indexes a cell in $A_{m_0}^{(j)}$ (but $K + n_j/4$ indexes a cell in $A_{m_1}^{(j)}$). Then, by Lemma 15, we have $\overleftarrow{D}(A_{m_1}^{(j)}) \leq n_j/12$. In this case, we need $D(A_{2i-1}^{(j+1)}) \leq 4\beta n_{j+1}$. Here, the dirtiness of $A_{2i-1}^{(j+1)}$ is determined by the number of 1's it contains, which is bounded by the intended number of 1's, which itself is bounded by $n_j/4 = n_{j+1}/2$, plus the number of 0's currently to the right of $A_{m_0}^{(j)}$, which, all together, is at most

$$n_{j+1}/2 + n_{j+1}/6 + \frac{2\beta n_{j+1}}{1-8\delta} + \frac{2\delta\beta n_{j+1}}{1-\delta} \leq 5n_{j+1}/6,$$

provided $\delta \leq 1/12$, $\epsilon \leq 1/32$, and $\beta \leq 1/180$. Thus, the Reduce algorithm, which is a $(\beta, 5/6)$-halver, we perform after the split gives $D(A_{2i-1}^{(j+1)}) \leq \beta n_{j+1}$.

4. $i = m_1$. In this case, there are subcases.
   (a) Suppose $K + n_j/4$ indexes a cell in $A_{m_0}^{(j)}$. Then, by Lemma 15, $\overleftarrow{D}(A_{m_1}^{(j)}) \leq \beta n_j$. In this case, we need $D(A_{2i-1}^{(j+1)}) \leq 4\beta n_{j+1}$, and $D(A_{2i}^{(j+1)}) \leq 4\beta n_{j+1}$, which follows from the above bound.
   (b) Suppose $K - n_j/4$ indexes a cell in $A_{m_1}^{(j)}$. Then, by Lemma 16, $\overleftarrow{D}(A_{m_0}^{(j)}) \leq 2\beta n_j$. In this case, m_0' and m_1' are both children of m_1, and the cross-over is in $A_{m_0'}^{(j+1)}$. We therefore need $A_{2i}^{(j+1)}$ to have dirtiness at most $n_{j+1}/6 = n_j/12$. The number of 0's in $A_i^{(j)}$ is at most $n_j/2$ (or this wouldn't be m_1), plus the number of additional 0's that are here because of 1's to the left of m_1, which is bounded by

$$n' = n_j/2 + \beta n_j + \frac{\beta n_j}{1-8\delta} + \frac{\delta\beta n_j}{1-\delta}.$$

If $c' \leq n_{j+1} = n_j/2$, then the ϵ-halver will reduce the dirtiness so that $D(A_{m_1'}^{(j+1)}) \leq \epsilon n_{j+1} \leq n_{j+1}/6$, if $\epsilon \leq 1/6$. If, on the other hand, $n' > n_{j+1}$, then, by Lemma 2, $D(A_{m_1'}^{(j+1)})$ will be reduced to be at most

$$\epsilon n_{j+1} + (1-\epsilon)\cdot(n' - n_{j+1}) \leq \frac{n_{j+1}}{6},$$

provided $\delta \leq 1/12$, $\epsilon \leq 1/32$, and $\beta \leq 1/180$.
   (c) Suppose neither of the previous subcases hold. Then we have two possibilities:
      i. Suppose K indexes a cell in $A_{m_0}^{(j)}$. Then $\overleftarrow{D}(A_{m_1}^{(j)}) \leq n_j/12$, by Lemma 15. We need $D(A_{2i-1}^{(j+1)}) \leq n_{j+1}/6$, which follows immediately from this bound, and we also need $D(A_{2i}^{(j+1)}) \leq 4\beta n_{j+1}$, which follows by our performing a Reduce step after we do our split.
      ii. Suppose K indexes a cell in $A_{m_1}^{(j)}$. Then, by Lemma 16, $\overleftarrow{D}(A_{m_0}^{(j)}) \leq n_j/12$. We need $D(A_{2i}^{(j+1)}) \leq 4\beta n_{j+1}$. Here, the dirtiness of $A_{2i-1}^{(j+1)}$ is determined by the number of 0's it contains, which is bounded by the proper number of 0's, which is at most $n_j/4 = n_{j+1}/2$, plus the number of 1's currently to the left of $A_{m_1}^{(j)}$, which, all together, is at most

$$n_{j+1}/2 + n_{j+1}/6 + \frac{2\beta n_{j+1}}{1-8\delta} + \frac{2\delta\beta n_{j+1}}{1-\delta} \leq 5n_j/6,$$

provided $\delta \leq 1/12$, $\epsilon \leq 1/32$, and $\beta \leq 1/180$. Thus, the Reduce algorithm we perform after the split will give us $D(A_{2i-1}^{(j+1)}) \leq \beta n_{j+1}$.

□

Putting everything together, we establish the following.

Theorem 6: *If it is implemented using a linear-time α-halver, Halver, for $\alpha \leq 1/15$, Zig-zag Sort correctly sorts an array of n comparable items in $O(n \log n)$ time.*

PROOF. Take $\alpha \leq 1/15$, for Halver being an α-halver, so that Reduce is simultaneously an ϵ-halver, a $(\beta, 5/6)$-halver, and a $(\delta, 5/6)$-attenuator, for $\delta \leq 1/12$, $\epsilon \leq 1/32$, and $\beta \leq 1/180$. Such bounds achieve the necessary constraints for Lemmas 8 to 18, given above, which establishes the dirtiness invariants for each iteration of Zig-zag Sort. The correctness follows, then, by noticing that satisfying the dirtiness invariant after the last iteration of Zig-zag Sort implies that the array A is sorted. □